The Paradis-Net API
(Technical Report)

Guido Malpohl and Florin Isailă

Institute for Program Structures and Data Organization, University of Karlsruhe,
76128 Karlsruhe, Germany Malpohl@ipd.uka.de, Florin@ipd.uka.de

Abstract. This paper describes the *Application Programming Interface* (API) of Paradis-Net, a typed event-driven message-passing interface for designing distributed systems. Paradis-Net facilitates the development of both peer-to-peer and client-server architectures through a mechanism called "Cooperation". We introduce the programming interface and demonstrate how the interface can be used to implement communication patterns typical for distributed systems.

1 Introduction

This technical report is an extension of the paper "Paradis-Net: A Network Interface for Parallel and Distributed Applications" [1]. Due to a very tight page limit it was not possible to publish an in-depth description of the API in the proceedings. The technical report gives these details and can be used as a reference by implementers using the library.

Paradis-Net is a typed message-passing interface for distributed applications and operating system services. Paradis-Net is suitable for designing distributed systems for both high speed network like Myrinet [2] or Infiniband [3] or for systems on top of relatively slow transport mediums like the Internet. A main goal of Paradis-Net is to offer a simple interface which facilitates the implementation of complex communication patterns and abstracts away from a particular high speed network.

Paradis-Net emerged from our experience in developing the Clusterfile parallel file system [4, 5]. In a typical parallel file system, the files are striped for performance reasons over the disks of several nodes. Therefore, the service of data requests or the update of the global metadata attributes involve the communication of one client with several servers. However, this communication pattern is not specific to parallel file systems. For instance, it can be used in DSM system to implement protocols for updating or invalidating copies of the same page residing on several nodes.

An other motivation for Paradis-Net was given by xFS [6], one of the first distributed decentralized file systems. The storage, cache and metadata are distributed over several workstations in order to provide file system services to the applications. The underlying protocols involve the cooperation of peers, trading off the simplicity of client-server model for the efficiency of a global cooperative

management policy. In an paper describing their experience with xFS [7], the authors identify the mismatch between the service they are providing and the available interfaces as a main source of implementation difficulties. We believe that Paradis-Net bridges this gap and demonstrate possible uses in section 3.

2 The Paradis-Net Architecture

This section introduces the Paradis-Net architecture by describing the available functions from the users point of view. Paradis-Net follows a peer-to-peer communication model in which every communication endpoint is a server and a client at the same time. The Paradis-Net library is a layer between the application and the native network interface. as depicted in figure 1. To the application it provides a simple, uniform interface independent of the actual network technology used, thus easing the development of complex distributed protocols. Paradis-Net can be extended with modules to support different network technologies and delivering their performance to the application.

After addressing the initialization procedure in section 2.1 and the communication primitive in section 2.2, we will discuss the server aspect of the interface in section 2.3 and the client aspect in section 2.4. To describe the API we will use ANSI C.

2.1 Initialization

Before any other function can be called, it is necessary to invoke *initialize* (see table 1). The function does not only initialize the internal data structures, but also opens the local endpoints. The argument contains the configuration options for the different network interfaces supported on this peer. Some implementations start a special servicing thread to manage incoming connection requests and messages. After the function returns, the peer can be contacted by others. The inverse operation *finalize* closes down all the endpoints, stops the servicing threads and releases the corresponding data structures.

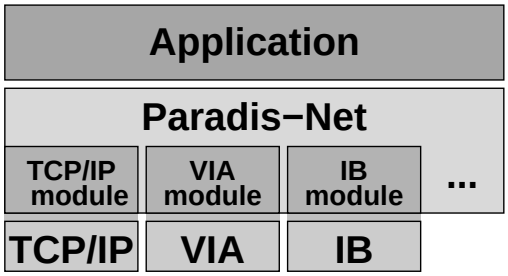

Fig. 1. Paradis-Net Architecture

General	
int initialize (*end_point* ep[])	section 2.1
void finalize ()	section 2.1
peer_id get_peer_id(**char** name[])	section 2.1
Communication	
int send(*peer_id* to, *msg_type* type, *coop_nr* nr, **void** *msg, **int** msg_size)	section 2.2
int forward(*peer_id* to, *peer_id* from, *msg_type* type, *coop_nr* nr, **void** *msg, **int** msg_size)	section 2.3
Request Handlers	
void set_handler(*msg_type* msg, **int** opt, *handler_fun* *handler)	section 2.3
void <*handler_fun*>(*peer_id* from, *msg_type* type, *coop_nr* nr, **void** * msg, **int** msg_size)	section 2.3
Cooperations	
coop_nr start_cooperation(*rcv_desc* *rcvec, **int** vec_size)	section 2.4
int end_cooperation(*coop_nr* nr, **int** timeout)	section 2.4
void cancel_cooperation(*coop_nr* nr)	section 2.4
int cooperation_finished(*coop_nr* nr)	section 2.4

Table 1. The Paradis-Net API

In Paradis-Net every endpoint has a unique peer name that can be expressed as an array of characters. This name usually consists of protocol and address information and is used to address other peers and to send messages to them. For example the name of a TCP endpoint is expressed as follows: *"tcp:<ip>:<port>"* where *<ip>* is the IP address and *<port>* the port number of the peer.

For convenience and performance reasons the peer names are mapped onto locally unique peer IDs by the function *get_peer_id* (see table 1). A peer ID is a small integer that is used to easier refer to remote endpoints. Using peer IDs is similar to using file handles instead of full file names in file system calls, but in contrast to file handles, peer IDs are not re-assigned at runtime.

2.2 Sending Data

The Paradis-Net library offers only one explicit communication primitive: *send* (see table 1). This operation sends a message to the peer represented by the peer ID (*to*). The type of the message is given as a parameter and will affect the way the message is handled on the receiving side. *send* will either return zero if the transfer was successful or an error number otherwise. After the operation returns the memory area that contains the message can be reused immediately. The *coop_nr* parameter can be disregarded for now, we will come back to it in section 2.4.

2.3 Request Handlers

Paradis-Net does not offer a function to receive data from other peers. Instead it uses handler functions that are called upon the arrival of a message. A handler function for a certain message type is set using the *set_handler* (see table 1) function. This event-driven mechanism implicates that the sending application and the receiving application have to agree on the meaning of the different message types at design time.

To avoid deadlock situations, Paradis-Net dispatches the up-calls to handlers in a dedicated thread. It allocates memory for each message on arrival and frees the memory as soon as the handler returns. In contrast to Active Messages, Paradis-Net handlers are not limited in their execution time and can initiate calls the library, including arbitrary *send* operations.

We will now look at the signature of a handler function (see table 1). The parameter *from* contains the ID of the peer that is the origin of the message that caused the invocation. The *type* field contains the type of the message, which is useful if a handler has been registered for several message types. The variable *msg* points to the received message and *msg_size* contains the size of that message. Again, *nr* has to be deferred until section 2.4.

Using handlers facilitates the implementation of server functionality: The reception of the request from the network and the invocation of the appropriate function to process it is being taken care of by the library. A handler function usually consists of the computation to fulfill the request and one call to *send* the reply back to the client, thus implementing the traditional client-server model.

The traditional model can be expanded in Paradis-Net with the *forward* (see table 1) function. The function, when called from within a handler, allows "forwarding" the message to a different peer and thereby also delegating the obligation to answer. The functions parameters are identical to the parameters of *send* with the exception of one addition parameter: *peer_id from*. When using this function, the handler on the next peer will not be invoked with the actual origin of the message, but with the local *peer_id* that corresponds to the *from* parameter of the *forward* call. Section 3.1 will give an example communication pattern the uses this operation.

2.4 Cooperations

While the handler concept eases the task of writing server functionality, it is not convenient for the implementation of client functionality: After a client sends a request to a server it has to wait for a reply before it continues. Although it is possible to use the handler mechanism of Paradis-Net to notify an application upon the arrival of a message, this would burden the application developer by having to implement additional synchronization.

For this reason Paradis-Net introduces "Cooperations". A Cooperation is a concept that defines a relationship between the outgoing requests and the incoming answers by creating a token that accompanies both messages. The

function *start_cooperation* (see table 1) has to be called to register a Cooperation on the client side. The parameter (*rcvec*) describes the reply that is expected.

As an introduction we will exemplify the life-cycle of a typical Cooperation in a client-server scenario: A Cooperation starts at the client. Before sending the request, the function *start_cooperation* registers the Cooperation and returns a token that represents the Cooperation. Next, the client will send a request to the server and afterwards call *end_cooperation* (see table 1). This function blocks until the expected result is available. The token accompanies the request message on its way to the server by using the optional parameter of the *send* operation (see table 1) that attaches the token to the message. On the servicing peer, Paradis-Net will invoke the handler that has been assigned to the message type with the Cooperation token as a parameter (see the signature of handler functions in table 1). When the reply is send, the Cooperation token is again attached to the message and travels back to its origin. This reply message has a special type which the client has assigned to be used for Cooperation replies. On the client site, Paradis-Net is therefore able to identify the reply as being part of a Cooperation. Although there is still the possibility to invoke a handler function upon the arrival of such a message, the library will first check if the attached cooperation token matches any of the currently active Cooperations on this peer. If this is the case, the service thread will store the message at the memory location that was declared when calling *start_cooperation* and afterwards wake up the thread that is waiting for the cooperation to finish.

After this introductory example, the following subsections will describe the operations involved in managing Cooperations in more detail to illustrate the additional possibilities they offer to implement advanced communication patterns.

Starting A Cooperation. We will first take a look at the *start_cooperation* operation and its parameter which is an array of receive descriptors (*rcv_desc*) that specify the expected reply. An array of descriptors is used, since the reply can consist of several messages whereby every element in the array represents one message from one peer. A Cooperation is completed when all elements have been filled with replies.

void∗ memory	Pointer to allocated memory
int size	Size of the allocated memory
int options	Selection criteria options
peer_id from	Sel. criteria: Sender peer ID
msg_type type	Sel. criteria: Message type
int received	Size of received message

Table 2. The Receive Descriptor

The fields of a receive descriptor are depicted in table 2. By assigning values to these fields, the user can influence (1) the selection of an array element at the reception of a message and (2) the memory handling. The field *receive* is set by Paradis-Net after receiving a message and storing it in this descriptor.
(1) When a reply arrives at the peer, the service thread locates the associated Cooperation to store the message in one of the receive descriptors. It selects a descriptor by iterating through the elements of the array and choosing the first element whose selection criteria fit the incoming message. This selection process is based on two features of the incoming message: The message type and the sender. By setting the variable *from* it is possible to restrict the receive descriptor to only accept replies from the peer with ID *from*. *type* can be used to confine a descriptor to a message type. Both selection criteria can be ignored by setting the option *RCV_FROM_ANY* and *RCV_ANY_TYPE* respectively. If both options are set, Paradis-Net will ignore the values in *from* and *type* and simply use the first free receive descriptor in the array. The selection criteria are particularly important for multi party communication patterns in which the different parties play different roles.
(2) The memory to store the reply message can be provided by the application or automatically allocated by Paradis-Net. If the memory is supplied by the application, a pointer to the memory and the size of the memory has to be stored in the appropriate fields of the descriptor (*memory* and *size*). When receiving the message, Paradis-Net will save up to *size* bytes of the message at the given location and -if necessary- discard the rest. The number of bytes that were received will be stored in *received*. The application can also mandate the library to automatically aquire as much memory as necessary by assigning 0 to *size*. While application provided memory incurs less overhead, the automatic allocation is useful if the size of the reply is not know a priori.

Waiting On A Cooperation. After initializing the receive descriptors, registering the Cooperation and sending requests to other peers, the application can either wait for the reply or do some computation and fetch the reply to the request at a later point. The library offers 3 operations that make it possible for applications to implement different policies: *end_cooperation*, *cooperation_finished* and *cancel_cooperation* (see table 1).

The function *end_cooperation* is the standard way to wait for a reply. If the Cooperation is already completed, it will de-register the Cooperation and immediately return. If the Cooperation is not yet completed, the current thread will sleep until the Cooperation is finished. *end_cooperation* usually returns a 0 to the caller on successful completion, while a negative value indicates an error, like an expired timeout for example.

cooperation_finished is a nonblocking version of *end_cooperation* with the difference that it does not de-register the Cooperation if it is completed. The function returns 0 if the Cooperation is not finished and 1 otherwise.

cancel_cooperation de-registers a Cooperation independent of their current status. It is mostly used in error situations, for example after a timeout expired

when calling *end_cooperation*. It can also be called from a different thread to wake up threads that are waiting for this Cooperation to finish in *end_cooperation*. Message replies that arrive after a Cooperation was canceled are discarded by Paradis-Net.

Receiving A Reply. The peer applications have to agree on certain message types to be used as Cooperation replies. These message types can also be associated with handlers, but in addition to that the messages will be stored in one of the registered receive descriptors if the message carries a valid Cooperation token and an empty and suitable descriptor is available. The operation *set_handler* is used to declare a message type as Cooperation reply by setting the option *COOPERATION*.

3 Examples

In this section we will illustrate the application of Paradis-Net in parallel and distributed file systems. We will demonstrate the flexibility and simplicity of the library by implementing two typical communication patterns. These examples stem from our own experience developing the parallel file system *Clusterfile* [4] and the observations that were made building the distributed file system *xFS* [7].

3.1 Delegation.

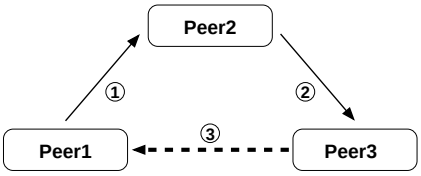

Fig. 2. Delegation in Paradis-Net

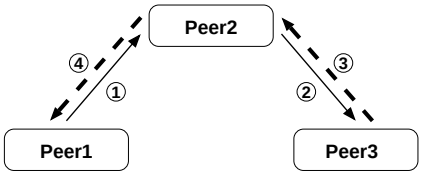

Fig. 3. Delegation through RPC

Solid arrow: Request – Dashed Arrow: Reply

The "delegation" pattern (Figure 2) has three participants: *Peer1* sends a request to *Peer2*. *Peer2* can not fulfill the request and forwards it to *Peer3*, which replies back to *Peer1*. The filesystem *xFS* uses this pattern for cooperative caching: A client (*Peer1*) reading from a file incurs a read miss in the local cache and sends a message to the cache manager (*Peer2*) in order to retrieve the cached data from a different peer. The manager consults its map, finds the responsible cache server (*Peer3*) and forwards the request to it. The cache server then responds back to the client with the cached data. The same pattern can be also be employed for routing in a peer-to-peer system.

Although this scenario appears to be simple, it is difficult to realize with traditional message passing interfaces. Wang et al. [7] demonstrate that RPCs are unsuitable to implement it because of the strict semantic imposed by the model. To synthesize the Delegation pattern, either 4 or 6 messages have to be send, even though only 3 messages are necessary (Figure 3).

We will now show code that implements the example. We use C syntax and for simplicity reasons, we assume that Paradis-Net has been initialized on all peers and the message types *REQUEST* and *REPLY* have been defined and registered with handlers and Cooperations respectively. When describing the implementation we will not detail on the precise content of the messages, instead we will focus on the intention and leave the details open.

Peer1 is the origin of the communication. The method *start* initiates it by registering a Cooperation, sending the request to a peer and waiting for an answer. According to figure 2, *start* is called with the peer ID of *Peer2*, the message and its length:

```
1  void start(peer_id to, void *msg, int msg_len) {
2    coop_nr coop;
3    rcv_desc desc = // receive descriptor
4      { memory: NULL, size: 0, type: REPLY,
5        options: RCV_FROM_ANY };
6
7    coop = start_cooperation(&desc, 1);
8    send(to, REQUEST, coop, msg, msg_len);
9     ... // eventual computation
10   end_cooperation(coop, 0); // no timeout
11 }
```

Listing 1. Sending the request

start first initializes the receive descriptor to accept messages with type *REPLY* from any peer (lines 3–5) and registers this Cooperation with the *start_cooperation* command (line 7). Next, the request is sent using the message type *REQUEST* and attaching the Cooperation token (line 8). Finally, the procedure will block in *end_cooperation* until the reply arrived (line 10). Note that the communication may overlap with compution between *send* and *end_cooperation* as the comment in line 9 suggests.

The *start* function is not customized for this example. It can be used for typical client-server communication as well. If -for example- *Peer2* would answer the request directly, *start* does not have to be changed at all, since the receive descriptor accepts replies from any peer, as long as the reply carries the Cooperation token issued by the local Paradis-Net library.

When *Peer2* receives the request from *Peer1*, Paradis-Net invokes the handler function *forward_handler*, which has been registered for messages with the type *REQUEST*. The handler first inspects the incoming message to find the peer responsible for answering the request and then forwards the message to it:

```
1  void forward_handler(peer_id from, msg_type type,
2                       coop_nr coop, void *msg,
3                       int msg_len) {
4    peer_id  liable_peer =
5                    find_liable_peer (msg, msg_len);
6
7    forward(liable_peer , from, msg_type, coop,
8            msg, msg_len);
9  }
```

Listing 2. Forwarding the request

The implementation of the function is straight-forward, it mainly consists of one call to forward the request to the liable peer. We abstract away from the application logic by expressing the finding of this peer as a function call. The implementation of *forward_handler* ignores errors that might happen when forwarding the message. In the case of an error, *Peer2* could reply back to *Peer1* with an error message, or try to forward the request to a different peer.

When *Peer2* sends the message to *Peer3*, the address of *Peer1* (the original source of the request) is piggy-backed on the message. On *Peer3* Paradis-Net calls the local handler function (*serve_request*) with the peer ID of *Peer1* as first parameter, so that the handler function will not be able to see the mediator that forwarded the request. After fulfilling the request, the handler will reply to the peer that was the origin of the request:

```
1  void serve_request(peer_id from, msg_type type,
2                     coop_nr coop, void *msg,
3                     int msg_len) {
4    reply_msg reply;
5    int reply_len;
6
7     fulfill_request  (msg, msg_len, &reply, &reply_len);
8
9    send(from, REPLY, coop, &reply, reply_len);
10 }
```

Listing 3. Serving the request

We hide the actual application logic behind a call to the *fulfill_request* method and assume that the result of this call will be stored in a *reply_msg* structure. When sending the reply (line 9), the parameter *from* is used as addressee, which is *Peer1* in our example. Again, we disregarded error handling: An error which occurs during the *send* operation will cause the reply not to be send.

3.2 Scatter/Gather.

Scatter/Gather is a 1-to-n communication pattern: *Peer1* requests the collaboration of a number of other peers to accomplish a task and contacts all peers

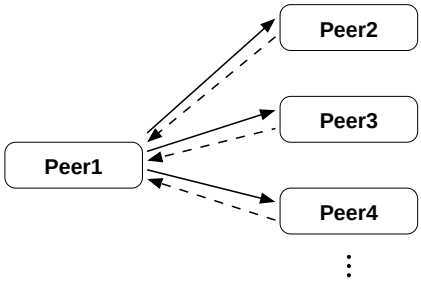

Fig. 4. The Scatter/Gather pattern
Solid arrow: Request – Dashed Arrow: Reply

in parallel. While the peers are processing, *Peer1* accepts the replies as soon as they arrive and continues with computation after all replies have been attained. Figure 4 illustrates the procedure.

We use this pattern in our parallel file system *Clusterfile* [4] when reading or writing data: Since files are striped over a number of data servers, it is necessary to contact several servers at the same time when accessing a file. Although the requests are send out in a particular order, the order of the replies is arbitrary.

For the following code, we again assume that Paradis-Net has been initialized on all peers and the message types *REQUEST* and *REPLY* have been defined and registered with handlers or Cooperations respectively.

The peers which play the server role in this pattern (*Peer2*, *Peer3*, ...), define a handler function to process the request. This handler will, after assembling an answer, reply back to *Peer1*. This procedure accords with the one of *Peer3* in the delegation example and therefore the implementation is the same: see listing 3.

As an extension of the pattern, it is also possible to forward the request to a different peer using the *forward* function, akin to listing 2. This would result in a combination of the Scatter/Gather and the Delegation pattern.

Before *Peer1* sends a request to the peers involved, it has to initialize receive descriptors and register them as a Cooperation. For simplicity reasons we send the same message to every peer and expect reply messages of type *reply_type*. After sending the requests, *Peer1* will block in *end_cooperation* until all answers have been received. The code is similar to the code of *Peer1* in the Delegation pattern (listing 1), except that there are a number of receive descriptors and *send* calls:

```
1 reply_type *start(peer_id* to, int n, void *msg,
2                   int msg_len) {
3   reply_type *replies =
4                   malloc(n * sizeof(reply_type));
5   rcv_desc *desc = init_desc(to, n, replies);
6   coop_nr coop;
7   int i;
```

```
8
9   coop = start_cooperation(desc, n);
10
11  for (i=0; i<n; i++)
12    send(to[i], REQUEST, coop, msg, msg_len);
13
14   ... // eventual computation
15
16  end_cooperation(coop, 0); // no timeout
17  free(desc);
18  return replies;
19 }
```

Listing 4. Sending requests

The parameters of the function *start* changed in comparison to listing 1 with regard to the addressee: Instead of passing just one peer ID, we pass all peer IDs of the hosts involved (*to*) as well as the number of hosts (*n*). The initialization of the receive descriptors is done in line 5 by the *init_desc* function. The function creates a vector of *n* descriptors to hold one reply each which is registered as a cooperation (line 9). All requests are send in succession by iterating over all peer IDs (lines 11–12). Registering all receive descriptors as one Cooperation allows us to wait for all replies concurrently by calling *end_cooperation* in line 16. After all replies are received, *end_cooperation* returns.

```
1  rcv_desc *init_desc(peer_id *to, int n,
2                      reply_type *replies) {
3    rcv_desc *desc = malloc(n * sizeof(rcv_desc));
4    int i;
5    for (i=0; i<n; i++) {
6      desc[i].memory = &replies[i];
7      desc[i].size = sizeof(reply_type);
8      desc[i].from = to[i];
9      desc[i].options = RCV_ANY_TYPE;
10   }
11   return desc;
12 }
```

Listing 5. Initializing the descriptors

We initialize the receive descriptors in such a way that each descriptor can only hold the reply from exactly one of the peers. We choose to disregard in the selection process, since all replies are of the same type. In listing 5 we use a simple *for* loop to do the initialization and again we ignore the possibility of errors occurring during the allocation of memory.

4 Conclusion

We have introduced Paradis-Net, a low-level network interface which is aimed the implementation of complex multi-party protocols. Paradis-Net emerged from our experience with a parallel file system and was motivated by the observation that the needs of our application were not satisfied by available network interfaces. Paradis-Net closes this gap with a simple design and the introduction of a collaboration mechanism called *Cooperation*.

Cooperations allow the user of Paradis-Net to describe the expected result of collaborative work between several participating peers. We described the mechanism and its potential use by demonstrating how two common communication patterns used in parallel file systems can be implemented.

We believe that Paradis-Net facilitates distributed system design and development of systems that follow the traditional client-server architecture as well as peer-to-peer or hybrid approaches.

References

1. Malpohl, G., Isailă, F.: Paradis-Net: A Network Interface for Parallel and Distributed Applications. In: Proceedings of the 4th International Conference on Networking. (2005)
2. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local area network. IEEE Micro **15** (1995) 29–36
3. InfiniBand Trade Association: InfiniBand Architecture Specification Release 1.1. (2002)
4. Isailă, F., Tichy, W.F.: Clusterfile: A flexible physical layout parallel file system. In: Proceedings of IEEE Cluster Computing Conference, Newport Beach. (2001)
5. Isailă, F., Malpohl, G., Olaru, V., Szeder, G., Tichy, W.F.: Integrating collective I/O and cooperative caching into the Clusterfile parallel file system. In: Proceedings of the ACM International Conference on Supercomputing (ICS). (2004)
6. Anderson, T.E., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., Wang, R.: Serverless network file systems. ACM Transactions on Computer Systems **14** (1996) 41–79
7. Wang, R.Y., Anderson, T.E.: Experience with a distributed file system implementation. Technical Report CSD-98-986, University of California at Berkeley (1998)